Security Audit – Neon EVM

Neodyme AG

November 5, 2021

Contents

Introduction **3**
Project Overview 3
Scope 4
Methodology 5

Findings **6**
Instruction Sysvar does not get checked (Critical) 7
Contracts can be deleted and modified (Critical) 8
External Calls execute Solana instruction instantly (Critical) 9
External calls always sign on behalf of the caller (Critical) 10
Denial of Service (High) 11
Infinite Execution Attack 11
Cancel-and-Refresh Attack 12
External Calls escalate operators signature (High) 13
CreateAccount allows to create different accounts with same Ethereum address (High) . . . 14
Transaction's target contract not verified (High) 15
ERC20 Wrapper allows privilege escalation (High) 16
Block of ERC20 Allowances and Ethereum addresses (Medium) 18
Nonce increment does not get written into account (Low) 19
Solana Chain Inconsistencies (Low) 20
Block of Contract Creation (Low) 21
Temporary Transaction Store can be overwritten (Low) 23
External calls allow reentrancy into the Neon program (Low) 24
Unused entrypoints available (Low) 25

Introduction

Neon engaged Neodyme to do a detailed security analysis of their on-chain programs. A thorough audit was done from September 13th to September 23rd.

The target of the audit was the end-to-end security from the moment a user signs an Ethereum transaction to execution in Neon on Solana. The audit revealed a a number of significant vulnerabilities as well as some medium and low-priority findings. All findings were subsequently fixed by the Neon team. The fix for a denial-of-service issue requires Neon's first MVP to rely on a trusted set of operators. These operators will be the only one allowed to forward Ethereum transactions into Neon, and will have full control over what transactions are executed and in what order.

The following report describes all findings in detail.

Project Overview

> Neon EVM is an Ethereum virtual machine on Solana that enables dApp developers to use Ethereum tooling to scale and get access to liquidity on Solana.

Executing Ethereum contracts on Solana is no easy task. Neon solves it by implementing a full Ethereum Virtual Machine (EVM) inside a Solana smart contract, alongside a range of storage and state handlers and helper functions. This raises many technical challenges due to the different chain-designs. As an example: on Ethereum, transactions get more expensive if they are more complex. On Solana, every transaction has a fixed complexity limit. This means a single Ethereum transaction might not fit into a single Solana transaction. Neon can therefore execute transactions iteratively over multiple Solana transactions. To make this transparent for users, a Neon Web3 Proxy exposes the usual Ethereum interface to users. It then does the iterative execution automatically. Neon operators run these proxies.

To take advantage of the broader Solana ecosystem, Neon implements an ERC20-Wrapper, which exposes an ERC-20-like interface for the Solana native SPL tokens. There are also various pre-compiled contracts, which allow interfacing with Solana from inside the EVM directly.

Scope

This audit looked at all on-chain and some off-chain parts of Neon. This enables Neodyme to evaluate the end-to-end security, from the moment the user signs an Ethereum transaction to execution in Neon on Solana. All audit targets are tagged with audit−20210913 by the Neon team. Specifically:

- Neon-EVM / evm_loader
  - The main Neon contract, which defines the interface between Solana and the EVM. Includes, for example, signature verification and state and storage handling. Also includes all ERC20Wrapper code.
  - github.com/neonlabsorg/neon-evm, (02c5944e855bf221f3f36eff95fc56f60cc0ff29)
- Rust EVM Implementation
  - The actual EVM implementation. Implements all EVM opcodes.
  - github.com/neonlabsorg/evm, (6076b6029b39d26e16180e910b49be3031950c5b)
- Neon Web3 Proxy:
  - Proxy run by operators, which convert Ethereum transactions into Neon transacations and execute them on Solana.
  - https://github.com/neonlabsorg/proxy-model.py/.../proxy/plugin/solana_rest_api.py
  - https://github.com/neonlabsorg/proxy-model.py/.../plugin/solana_rest_api_tools.py

Methodology

Neodyme's audit team performed a comprehensive examination of the Neon contract. The team, which consists of security engineers with extensive experience in Solana smart contract security, reviewed and tested the code of the on-chain contracts, paying particular attention to the following:

- Ruling out common classes of Solana contract vulnerabilities, such as:
  - Missing ownership checks,
  - Missing signer checks,
  - Signed invocation of unverified programs,
  - Solana account confusions,
  - Re-initiation with cross-instance confusion,
  - Missing freeze authority checks,
  - Insufficient SPL token account verification,
  - Missing rent exemption assertion,
  - Casting truncation,
  - Arithmetic over- or underflows,
  - Numerical precision errors.
- Checking for unsafe design, which might lead to common vulnerabilities being introduced in the future,
- Checking for any other, as-of-yet unknown classes of vulnerabilities arising from the structure of the Solana blockchain,
- Ensuring that the contract logic correctly implements the project specifications,
- Examining the code in detail for contract-specific low-level vulnerabilities,
- Ruling out denial-of-service attacks,
- Ruling out economic attacks,
- Checking for instructions that allow front-running or sandwiching attacks,
- Checking for rug-pull mechanisms or hidden backdoors.

Findings

All findings are classified in one of four severity levels:

- **Critical**: Bugs which will likely cause loss of funds. This means that an attacker can, with little or no preparation, trigger them, or they are expected to happen accidentally. Effects are difficult to undo after they are detected.
- **High**: Bugs, which can be used to set up loss of funds in a more limited capacity, or to render the contract unusable.
- **Medium**: Bugs that do not cause direct loss of funds but lead to other exploitable mechanisms.
- **Low**: Bugs that do not have a significant immediate impact and could be fixed easily after detection.

Name	Severity
Instruction Sysvar does not get checked	Critical
Contracts can be deleted and modified	Critical
External Calls execute Solana instruction instantly	Critical
External calls always sign on behalf of the caller	Critical
Denial of Service	High
External Calls escalate operators signature	High
CreateAccount allows to create different accounts with same Ethereum address	High
Transaction’s target contract not verified	High
ERC20 Wrapper allows privilege escalation	High
Block of ERC20 Allowances and Ethereum addresses	Medium
Nonce increment does not get written into account	Low
Solana Chain Inconsistencies	Low
Temporary Transaction Store can be overwritten	Low
Block of Contract Creation	Low
External calls allow reentrancy into the Neon program	Low
Unused Entrypoints Available	Low

Instruction Sysvar does not get checked (Critical)

Severity	Impact	Affected Component
Critical	Signature verification can be circumvented	Call Entrypoints

The instructions used to start processing an Ethereum transaction have a critical vulnerability caused by a missing account check. By supplying a maliciously crafted account, an attacker can bypass the signer check and execute any transaction in the name of any user. This impersonation will lead to loss of funds.

The bug is caused by a missing verification that the passed sysvar_info account matches the expected Sysvar :: Instructions sysvar. The sysvar is used in fn check_secp256k1_instruction() to verify that the Solana transaction contains a valid secp256k1-instruction. This secp256k1-instruction in turn verifies that the Ethereum caller signed the Ethereum transaction. If the signature is invalid, the whole transaction is terminated. An attacker can thus create an account that looks like the correct sysvar account, except that the verification has passed.

All Call instructions, except for ExecuteTrxFromAccountDataIterativeOrContinue are affected, because there the signature check is not done with a secp256k1 instruction, but via the sol_secp256k1_recover syscall.

Solution The Neon team added the sysvar account check in check_secp256k1_instruction. Neodyme verified the fix.

- https://github.com/neonlabsorg/neon-evm/issues/259
- https://github.com/neonlabsorg/neon-evm/commit/281cdd7a2590bdeef4401e36079a16aaa8f41d4f

Contracts can be deleted and modified (Critical)

Severity	Impact	Affected Component
Critical	Contract code can be modified by anyone	CreateAccount

The entrypoint used to create new Neon/Ethereum accounts has a critical vulnerability caused by a missing account check, allowing anyone to modify any deployed contract.

CreateAccount creates a new Neon/Ethereum account as a collection of multiple Solana accounts. When creating an account for a contract, one provides a program_code Solana account, which is used to store the contract's code.

Neon does not check if the program_code account is already in use. This allows an attacker to re-initialize an already existing program_code account of another Ethereum contract. These changes then get written back into the account through `contract_data.pack(&mut program_code.data.borrow_mut())?;`.

This effectively deletes the target Ethereum contract from the chain. Further, it is now possible to place new contract code in the re-initialized account.

Solution Neon now checks that the code account is zero-initialized. Neodyme verified this fix.

- https://github.com/neonlabsorg/neon-evm/issues/258
- https://github.com/neonlabsorg/neon-evm/pull/293

External Calls execute Solana instruction instantly (Critical)

Severity	Impact	Affected Component
Critical	Execution cannot be rollbacked	external_call

Neon has a built-in SYSTEM_ACCOUNT_SOLANA contract, that allows any EVM contract to call a native Solana contract. The actual call is implemented in account_storage.rs in the external_call function.

A critical issue occurs when combining three facts:

- A single Ethereum transaction can be iteratively executed in multiple Solana instructions.
- external_call executes instructions immediately via Solana cross-program-invocation (CPI).
- CPIs cannot be rolled-back once the transaction containing them has succeeded.

Take, for example, an Ethereum transaction that is split into two Solana transactions. The first succeeds and does an external call to a native contract. This contract is now executed immediately. In the second, the EVM reverts. But the external-call is already persisted on the Solana chain and can not be reverted.

A malicious operator can intentionally split the execution of an Ethereum transaction so that some native call succeeds while the main Ethereum transaction is reverted.

Solution The Neon team won't include this functionality in the initial release. Neodyme verified this fix.

- https://github.com/neonlabsorg/neon-evm/issues/269
- https://github.com/neonlabsorg/neon-evm/pull/271

External calls always sign on behalf of the caller (Critical)

Severity	Impact	Affected Component
Critical	Contract can steal all of users/parent contract's Eth/ERC20 tokens	external_call

Neon has a built-in SYSTEM_ACCOUNT_SOLANA contract, that allows any EVM contract to call a native Solana contract. The actual call is implemented in account_storage.rs in the external_call function. The function calls invoke_signed on the passed instruction and signs with seeds of both the instruction caller and the called contract:

```
invoke_signed(
    instruction,
    account_infos,
    &[&sender_seeds[..], &contract_seeds[..]]
)
```

The used seeds are the same ones as used for the authority of the SPL token wallets, which contain the callers/contract's ERC20 and Ethereum balances. It is thus possible for an Ethereum contract to do a native Solana SPL token transfer, signed with the caller's seeds. This allows the contract to withdraw an arbitrary amount from any of the user's wallets.

This breaks a security premise of Ethereum, as a transaction usually has a specified value of Wei, that get transferred from the caller to the recipient of the transaction. It must not be possible for a contract to receive more than the user specified.

In addition, the contract seeds are always taken from the first, user-called contract. They are not updated when a call is made inside Ethereum. This means that any child contract can use the same authority as the parent contract.

Solution The Neon team won't include this functionality in the initial release. Neodyme verified this fix.

- https://github.com/neonlabsorg/neon-evm/issues/269
- https://github.com/neonlabsorg/neon-evm/pull/271

Denial of Service (High)

Severity	Impact	Affected Component
High	Denial of Service	Iterative Execution

The Neon contract in its current version is vulnerable to denial-of-service attacks, where arbitrary Ethereum accounts can be blocked from being acted upon. This can be used to block certain users or contracts.

On Ethereum, there is a mempool for transactions. This allows miners to only execute profitable instructions, and makes a denial-of-service attack prohibitively expensive. An attacker who wants to prevent a specific contract from being executed would have to "buy" every single execution slot. Using smart execution ordering, arbitrageurs can sandwich swap instructions and earn the maximum slippage, but not much more.

Neon does not have such a pool, and the current implementation is first-come-first-serve.

Infinite Execution Attack

Let us assume an attacker wants to block a contract X from executing. Such a contract could, for example, be a liquidity pool, where freezing of price could potentially have a significant impact.

The attacker could:

1. Write & deploy his own contract Y to Neon, which just enters an infinite loop.
2. He creates & signs a transaction T contract Y. As Neon has no global gas-price, he just chooses 1 Wei (maybe even 0). Since gas is so cheap, he can use a gas_limit of u64::max.
3. He uploads the instruction to Neon, and calls ExecuteTrxFromAccountDataIterative to begin executing it iteratively, choosing himself as operator. As he wants to block contract X with this transaction, he provides its account in trx_accounts, which means it will get blocked, and no other transaction can use it until this one is complete or canceled.
4. He then calls Continue once per slot, with step_count=1.
5. If no other operator steps in, he can continue to do so indefinitely. The earliest another operator can cancel is after the OPERATOR_PRIORITY_SLOTS are over, or about 8 seconds.

Conclusion Operators need to look for this kind of blocked transactions and not only try to progress them (as the gas limit is too high), but actively cancel them when it is lucrative, even if some other operator is currently working on them.

That in itself is not a significant issue, just something smart operators will do anyway. To prevent the attacker from getting a foothold again, the operators can even place the cancel and begin_new_tx instructions in the same Solana transaction. But that is something an attacker can do as well, in a cancel-and-refresh attack.

Cancel-and-Refresh Attack

Shortly before the priority slots of the attacker run out, he cancels his own Neon transaction, and in the same Solana transaction begins it again.

This process can currently be continued indefinitely. The priority slots will never run out, as the attacker constantly renews them. The attacker could even cycle through operator identities to hide this a bit. This can block both, contract execution and individual users.

In the context of DeFi swaps this can be leveraged to sandwich transactions with large slippage by blocking the swap contract until a transaction gets uploaded in preparation of execution. Since the attacker blocks the pool, he can sandwich this victim transaction with near certainty.

Solution The first release of Neon will only allow permissioned operators to make transactions. This means that a user will have trust all operators, as any single of of them could still execute the attack above, at least until they are manually removed from the operator set. It is also possible for operators to selectively drop transactions they receive, or sandwhich them as they choose. Neodyme has verified this partial fix.

- https://github.com/neonlabsorg/neon-evm/pull/308

External Calls escalate operators signature (High)

Severity	Impact	Affected Component
High	Loss of Operator Funds	external_call

Neon has a built-in SYSTEM_ACCOUNT_SOLANA contract, that allows any EVM contract to call a native Solana contract. The actual call is implemented in account_storage.rs in the external_call function. The function calls invoke_signed on the passed instruction:

```
invoke_signed(
    instruction,
    account_infos,
    &[&sender_seeds[..], &contract_seeds[..]]
)
```

In such a cross-program invocation, all signers of the parent instruction can be available to the called contract. In the case of Neon, all Solana transactions are paid by an operator, whose fee wallet therefore has to sign. A malicious Ethereum contract could specify the Operators fee wallet as signer in the CPI call, which the Solana runtime will treat as valid. Thus, the Operators signature is escalated and can be used on an arbitrary call.

Example Attack Scenario:

- Attacker deploys a Solana contract, that transfers all funds from all signing wallets to his own wallet.
- Attacker deploys an Eth contract that makes an external call to the Solana contract, specifying the operator's fee wallet as signer.
- The attacker sends an Eth instruction which calls his Eth contract to the victim operator.
- The operator will sign the Solana transaction containing the eth transaction and execute it.
- All funds from the operator's fee wallet are stolen into the attacker's wallet.

It is important to note that using invoke will not fix this issue, as invoke internally calls invoke_signed without signer_seeds. Instead, Neon has to manually verify that the CPI instruction does not specify the operator as a signer.

Solution The Neon team won't include this functionality in the initial release. Neodyme verified this fix.

- https://github.com/neonlabsorg/neon-evm/issues/269
- https://github.com/neonlabsorg/neon-evm/pull/271

CreateAccount allows to create different accounts with same Ethereum address (High)

Severity	Impact	Affected Component
High	Transaction Replay / Contract Mutability	CreateAccount

Accounts are created as program-derived-addresses (PDAs). They have a seed chosen to ensure the PDA can not have a standard private key. Neon does not check that the seed used for the provided PDA account is the first one. This allows an attacker to create separate Neon accounts for the same Ethereum address.

This new account will have its own Ethereum balance and nonce. Because the nonce is reset, old transactions can be replayed. In order for the nonce to match, they have to be replayed in-order. But as transactions can be started and immediately cancelled, it is possible to fast-forward the nonce without actually executing all old transactions. Old transactions are not persisted on-chain directly, but will be available to query via RPC, so the attacker does not have to actively observe an account from the beginning.

A small limitation exists, because the new Neon account will have its own Eth balance. This means accessing the original user's Eth funds directly is impossible, and the attacker will have to provide the necessary Eth funds for all replayed transactions. But because everything, except for the native Eth balance is shared, there will be many transactions, where the value of the transferred Eth is lower than the value generated by the instuction. This is for example the case for most ERC20 transfers.

Another attack-vector possible with this bug, is to deploy two different contracts at the same Ethereum address! Regular users will always use the "main" contract at the lowest seed, since this is the one all normal operators will derive. But the attacker can capture selected transactions and provide Neon with his duplicated account, which contains different code. This potentially malicious "shadow" contract has the same authority because it runs at the same address.

Solution Neon now verifies that the seed is the first one, which is the one returned by fn Pubkey::find_program_address(). Neodyme verified this fix.

- https://github.com/neonlabsorg/neon-evm/issues/264

Transaction's target contract not verified (High)

Severity	Impact	Affected Component
High	Loss of Funds	Call Entrypoints

The Call entrypoints receive an Ethereum transaction, which is signed for a specific contract (trx.to). Neon does not verify that the given contract-account matches the trx.to address. Therefore a malicious operator could take the signed transaction and execute it against a different contract.

This will allow anyone who knows a not-yet-executed, but signed, user transaction to steal all Eth tokens transferred in it. This can be done by calling the transaction against an attacker deployed contract, that does nothing except receiving funds without aborting.

Additionally, this attack is interesting for ERC20 contracts, where a transfer of Token X could be turned into a transfer for Token Y, but with the same recipient and token-amount.

This attack can be trivially executed the user's chosen operator, but also by an outside attacker that is able to intercept the user's transaction while it is uploaded to Solana.

Solution The Neon team implemented a check in check_ethereum_authority. Neodyme verified this fix.

- https://github.com/neonlabsorg/neon-evm/issues/262

ERC20 Wrapper allows privilege escalation (High)

Severity	Impact	Affected Component
High	Contract can steal all of users Eth/ERC20 tokens	ERC20Wrapper

The current implementation of the ERC-20 Wrapper has a severe bug, which allows a contract called by a user to steal all ERC20 and Eth tokens from said user.

The goal of the wrapper is to expose the native Solana SPL tokens via an ERC-20-like interface to contracts. In particular, this means providing a `transfer(address _to, uint256 _value)` function.

On Ethereum, one user-account can have many different ERC-20 tokens simply by holding their balances in the ERC-20 contracts. The user's address is used as an identity to receive tokens, and as an authority to transfer them.

The same is possible on Solana via associated-token-accounts (ATAs). A user's public-key is used to derive addresses to token accounts for specific mints. This means that two pieces of information are necessary to transfer SPL tokens to a user: the user's public-key, and the mint public-key. On Ethereum, the situation is similar. You need the user's public-key and the ERC-20 contract address.

The ERC20-Wrapper now has the task to convert the mint address into an ERC-20 contract address and vice versa. It does so by forwarding all calls it receives to Neons pre-compiled SYSTEM_ACCOUNT_ERC20_WRAPPER contract via a delegate-call while injecting the Solana mint address into the arguments. The Solana mint address is stored in the wrapper's contract storage when the wrapper is deployed.

Because the wrapper is not a privileged contract in any way, any contract can pretend to be the wrapper for many different SPL tokens. A malicious contract can thus transfer arbitrary ERC-20 tokens from any user that calls it.

From a technical perspective, this is caused by Neon's ERC-20 system-call blindly trusting the caller as the source of the transaction. The wrapper uses a delegate-call, in which the caller is not changed, so a user can transparently access his own SPL account and not the wrappers. Any contract can `delegate_call` into the ERC-20 system-call, and the caller for the original contract is seen as trusted. On real Ethereum, a `delegate_call` to an ERC-20 contract would do nothing since the actual storage of said ERC-20 contract can not be modified when a delegate call is used.

Another interesting, though not necessarily problematic, fact about this implementation: Neon's Ethereum tokens are also represented as SPL tokens and use the same authority as ERC-20 tokens. It is thus possible to transfer Eth via ERC-20 calls and also to create allowances for Eth.

Solution Instead of using associated-token-addresses for the Ethereum user's SPL token accounts, Neon now uses custom-derived accounts. This means each wrapper now defines its own distinct ERC-20 token, even if the underlying SPL token is the same. These separate token accounts for each ERC20 contract are created via EvmInstruction :: ERC20CreateTokenAccount, with the wallet address now being dependent on the user, the SPL-mint, and also the Ethereum wrapper address. Neodyme has verified the fix.

- https://github.com/neonlabsorg/neon-evm/issues/279
- https://github.com/neonlabsorg/neon-evm/pull/291

Block of ERC20 Allowances and Ethereum addresses (Medium)

Severity	Impact	Affected Component
Medium	Partial Denial of Service	Account Creation

The Solana accounts for ERC20 Allowances and Ethereum accounts are created through the Solana syscall SystemInstruction :: CreateAccount. This is an issue since it will fail if the account already exists, with existing being defined via the account balance: to.lamports() > 0. Everybody can transfer lamports to any address, so an attacker could block specific addresses from creation by transferring some lamports to it.

Most addresses Neon uses are deterministically computed from seeds. For example, an attacker can block the creation of a specific Ethereum account in EvmInstruction :: CreateAccount.

In the context of Neon ERC20 Allowances, an account derived from (mint, owner, spender) is created. An attacker can block arbitrary of these tuples, preventing the creation of allowances for a specific ERC20/owner/spender instance. This can be a big issue if contracts use a pull-based model to transfer funds. For example, a user could be blocked from withdrawing funds from a swap.

Solution The Neon now provides implements the create_pda_account function, which checks if the account already has lamports and if so transfers, allocates and assigns manually. Neodyme verified this fix.

- https://github.com/neonlabsorg/neon-evm/issues/278

Nonce increment does not get written into account (Low)

Severity	Impact	Affected Component
Low	Cancelled TX can be restarted	Cancel Entrypoint

The Cancel instruction is used to cancel a pending instruction. This can be done either by the original operator or by another operator after the priority period has passed. To ensure a canceled instruction stays canceled, the nonce of the caller account is increased. However, this change is not written back into the account using AccountData::pack and therefore does not persist, making it possible to start the same transaction again.

Solution Neon now correctly writes the nonce-increase into the account. Neodyme verified the fix.

- https://github.com/neonlabsorg/neon-evm/issues/295
- https://github.com/neonlabsorg/neon-evm/pull/326

Solana Chain Inconsistencies (Low)

Severity	Impact	Affected Component
Low	Etherum contracts might observe inconsistent state	Solana Interactions

A contract running inside Neon has multiple avenues to read data from the Solana blockchain. These values are not always constant during the iterative execution of a longer contract. The current implementations do not always cache the data, making it possible that the same call returns different values in a single Ethereum transaction. This makes Time-of-Check – Time-of-Use (TOCTOU) bugs possible.

Any value that is read from Solana, which is not controlled by Neon alone, should be cached.

More specifically, the values of the current block, block_time and all SPL (and ERC20) token balances can change:

- The erc20-wrapper syscall can provide total_supply and balances of SPL token accounts, which are changeable between iterations of neons iterative executions.
- Everything that uses apply_to_solana_account is affected.
- A similar bug also affects the re-initialization of the ProgramAccountStorage when execution resumes. All SPL balances are read fresh. This is an issue if (ERC-20) tokens are traded without Neon on native Solana, though balances can only increase because, inside Neon, the account is blocked so that it won't sign a transfer instruction.

Solution Neon now caches all values read from Solana, which are not controlled by Neon alone. Neodyme verified the fix.

- https://github.com/neonlabsorg/neon-evm/issues/296
- https://github.com/neonlabsorg/neon-evm/pull/318

Block of Contract Creation (Low)

Severity	Impact	Affected Component
Low	Block a contract from deploying another contract	Account Creation

When creating an Ethereum account via EvmInstruction :: CreateAccount, the caller freely chooses if the target account should be a user-account or a code-account. The two types are identical, except that the code-account stores the address of a provided program_code account, which will be used to store the contract's code.

The user/code choice is currently final, as there is no way to "upgrade" a user-account into a code-account.

Now consider the CREATE opcode in Ethereum. It allows a contract to deploy another new contract. This new contract's address will be derived only from the old contract's address and nonce.

This information is known to everyone. An attacker could thus create the account where the next contract would be deployed at as a user-account.

The targeted contract now could never use the CREATE opcode, as Neon can not allow creating a contract in a user-account.

CREATE2 can be broken the same way, but there you cannot block all future calls by creating a single eth-account. Instead, you have to actively block the eth-account for each salt the contract is expected to supply to create2 individually.

This is exactly the use case CREATE2 is intended for:

> Allows interactions to (actually or counterfactually in channels) be made with addresses that do not exist yet on-chain but can be relied on to only possibly eventually contain code that has been created by a particular piece of init code. Important for state-channel use cases that involve counterfactual interactions with contracts.

Overall, this bug has a low impact, as Neon can be upgraded to allow user-to-contract account upgrades when this issue occurs, and nothing is permanently blocked.

Solution The Neon team now allows resizing existing accounts. In the same vein, they allow to change a user account into a code account, but only if the user account has not made a transaction yet. It might still be possible to trip up automated contract-deploy systems, but it can always be fixed manually and the deploy system improved to handle this attack correctly. Neodyme verified the fix.

- https://github.com/neonlabsorg/neon-evm/issues/355
- https://github.com/neonlabsorg/neon-evm/pull/356
- https://github.com/ethereum/EIPs/blob/master/EIPS/eip-1014.md

Temporary Transaction Store can be overwritten (Low)

Severity	Impact	Affected Component
Low	Operator operation can be disturbed	Write Entrypoint

The current architecture of EvmInstruction :: Write allows anyone to write into any empty code account and accounts of type AccountData::Empty.

This is an issue, as operators currently use these empty data accounts to temporarily store bigger transactions before they can begin executing them. An attacker can partially overwrite these accounts, invalidating the Ethereum signature and preventing the TX from being executed. A sophisticated attacker could use this to block an operator from executing any large transaction.

Solution Neon now only allows writes into accounts created from the operators pubkey and a seed with Pubkey::create_with_seed, where the operator has to sign. It is thus no longer possible to write into accounts of other operators. Neodyme has verified the fix.

- https://github.com/neonlabsorg/neon-evm/issues/261
- https://github.com/neonlabsorg/neon-evm/pull/287

External calls allow reentrancy into the Neon program (Low)

Severity	Impact	Affected Component
Low	Could break some primitives	external_call

Neon has a built-in SYSTEM_ACCOUNT_SOLANA contract that allows any EVM contract to call a native Solana contract. The actual call is implemented in account_storage.rs in the external_call function.

Currently, this also allows to call the Neon contract itself again. While we do not see any immediate security implications, this kind of re-entrancy can be dangerous. If there isn't a good reason to allow this, we recommend blocking Neon from calling itself.

One example where this could be an issue is clients that consume the OnEvent and OnReturn CPI calls. These might now occur multiple times on different call-stack-depths, which clients likely won't expect. This can cause clients to miss important events.

Solution The Neon team won't include this functionality in the initial release. Neodyme verified this fix.

- https://github.com/neonlabsorg/neon-evm/issues/269
- https://github.com/neonlabsorg/neon-evm/pull/271

Unused entrypoints available (Low)

Severity	Impact	Affected Component
Low	Increased Attack Surface	Entrypoint

There are multiple entrypoints defined, which are not expected to be used, specifically Finalize and CreateAccountWithSeed. Removing these functions will reduce Neon's attack surface.

Solution The functions are leftover from a previous iteration and are now removed. Neodyme verified the fix.

- https://github.com/neonlabsorg/neon-evm/issues/263
- https://github.com/neonlabsorg/neon-evm/issues/265

Neodyme AG

Dirnismaning 55
Halle 13
85748 Garching
E-Mail: contact@neodyme.io

https://neodyme.io